

Developing Embedded Linux by Poky

HAYASHI Kazuhiro

Advanced Software Technology Group
Corporate Software Engineering Center
TOSHIBA CORPORATION

2012/09/20

はじめに

■ 組込み向けLinux開発における要求・課題

- クロス開発環境の準備
- プログラム(パッケージ)間の問題解決
 - 依存、競合、バージョンごとの互換性
- 最小構成の実現
 - 必須機能以外を省いた環境構築
- 多重開発の防止
 - 共通化可能なリソースの発見と流用



■ 組込み向けLinuxビルドツール「Poky」を活用

- Pokyの概要説明
- 自分向けにカスタマイズする
- 使用上の課題を考える

対象バージョン:
Poky 6.0 edison (Yocto Project 1.1)

目次

■ Pokyの概要

- 全体像
- 構成要素
- ビルドフロー
- ディレクトリ構造

■ レシピの作成

- 記述方法
- 例)helloレシピ

■ Pokyをカスタマイズする

- バージョンアップ
- カーネルレシピの作成
- BSPの作成

■ 使用上の課題

目次

■ Pokyの概要

- 全体像
- 構成要素
- ビルドフロー
- ディレクトリ構造

■ レシピの作成

- 記述方法
- 例) helloレシピ

■ Pokyをカスタマイズする

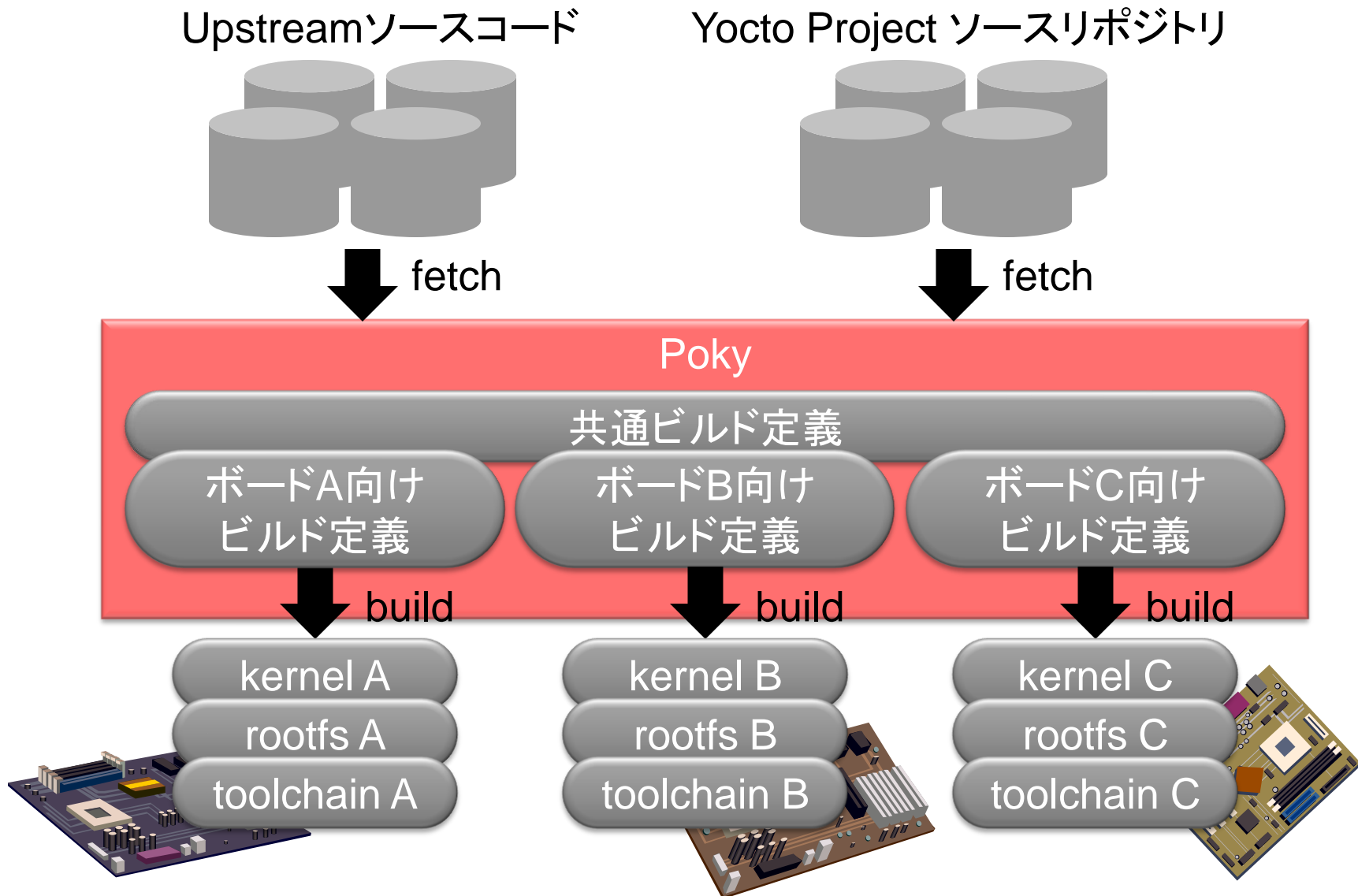
- バージョンアップ
- カーネルレシピの作成
- BSPの作成

■ 使用上の課題

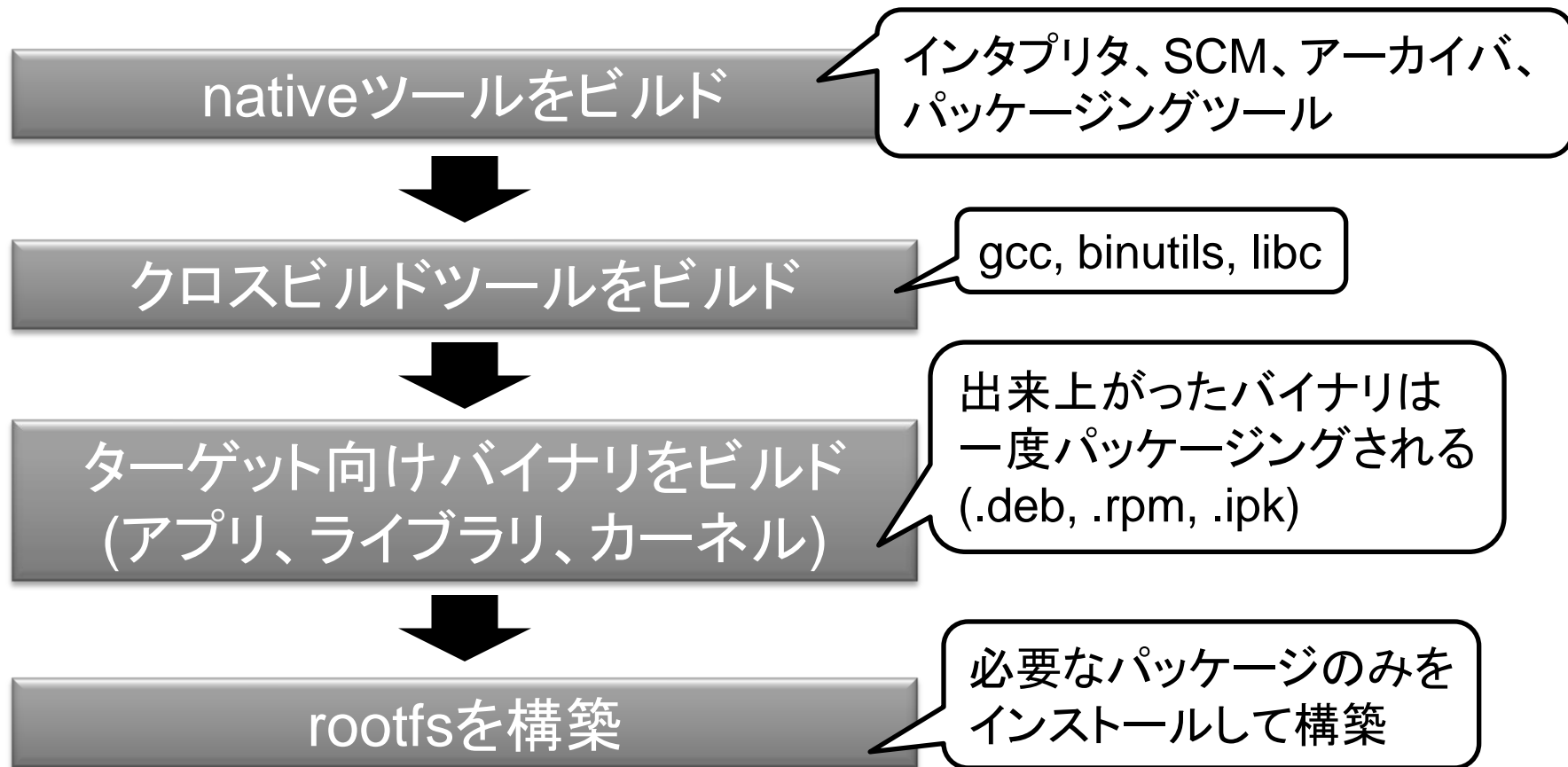
Pokyの概要

- **組込み向けカスタムLinuxを構築可能なビルドシステム**
 - 出力:カーネル、ユーザランド(rootfs)、ツールチェーン
 - パッケージ単位のカスタマイズが容易
 - Yocto Projectの主要コンポーネントの一つ
- **Yocto Project (<http://www.yoctoproject.org/>)**
 - 幅広いハードウェア向けにカスタムLinuxディストリビューションを構築するための開発基盤・ツールを提供
 - 開発者の負荷を軽減、開発の複雑性を緩和
- **対応アーキテクチャ**
 - x86(32bit & 64bit), ARM, PowerPC, MIPS
- **GUIによる操作**
 - ビルド対象の選択・カスタマイズやビルドシーケンスの確認
 - Eclipse経由でアプリケーション開発、デバッグが可能

全体像



ビルドの手順



ターゲット向けバイナリの生成に必要なツールは、基本的にnative環境に依存しない

主な構成要素

■ レシピファイル (*.bb)

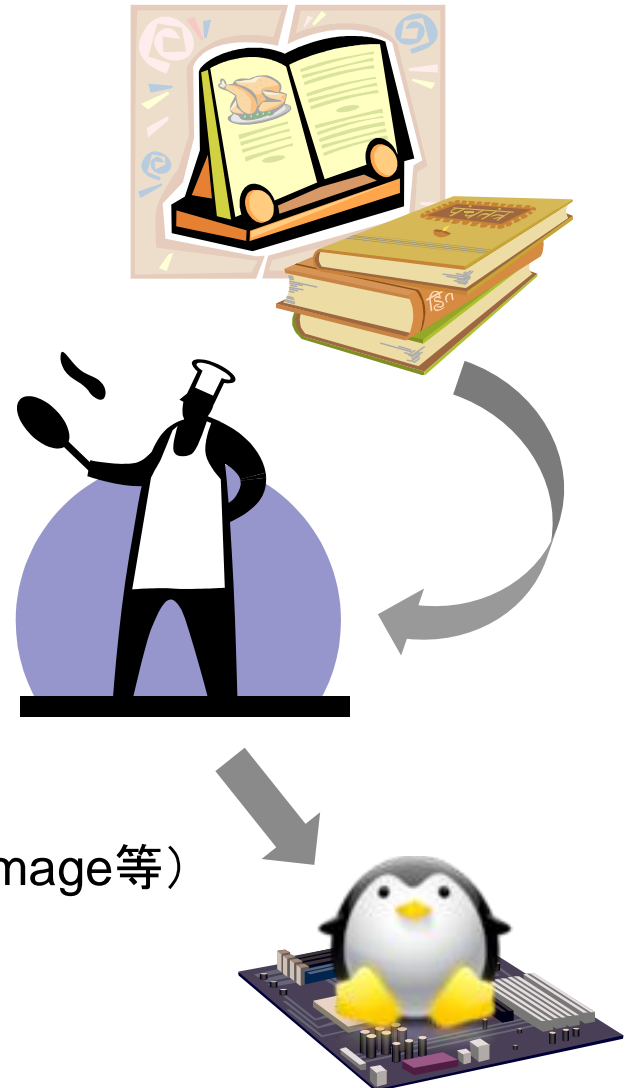
- ビルド方法の定義書(テキスト)
- シェルスクリプト+Pythonの混合記述
- ソースfetch～パッケージングまでの動作を定義
- レシピ同士の依存関係を定義

■ bitbake

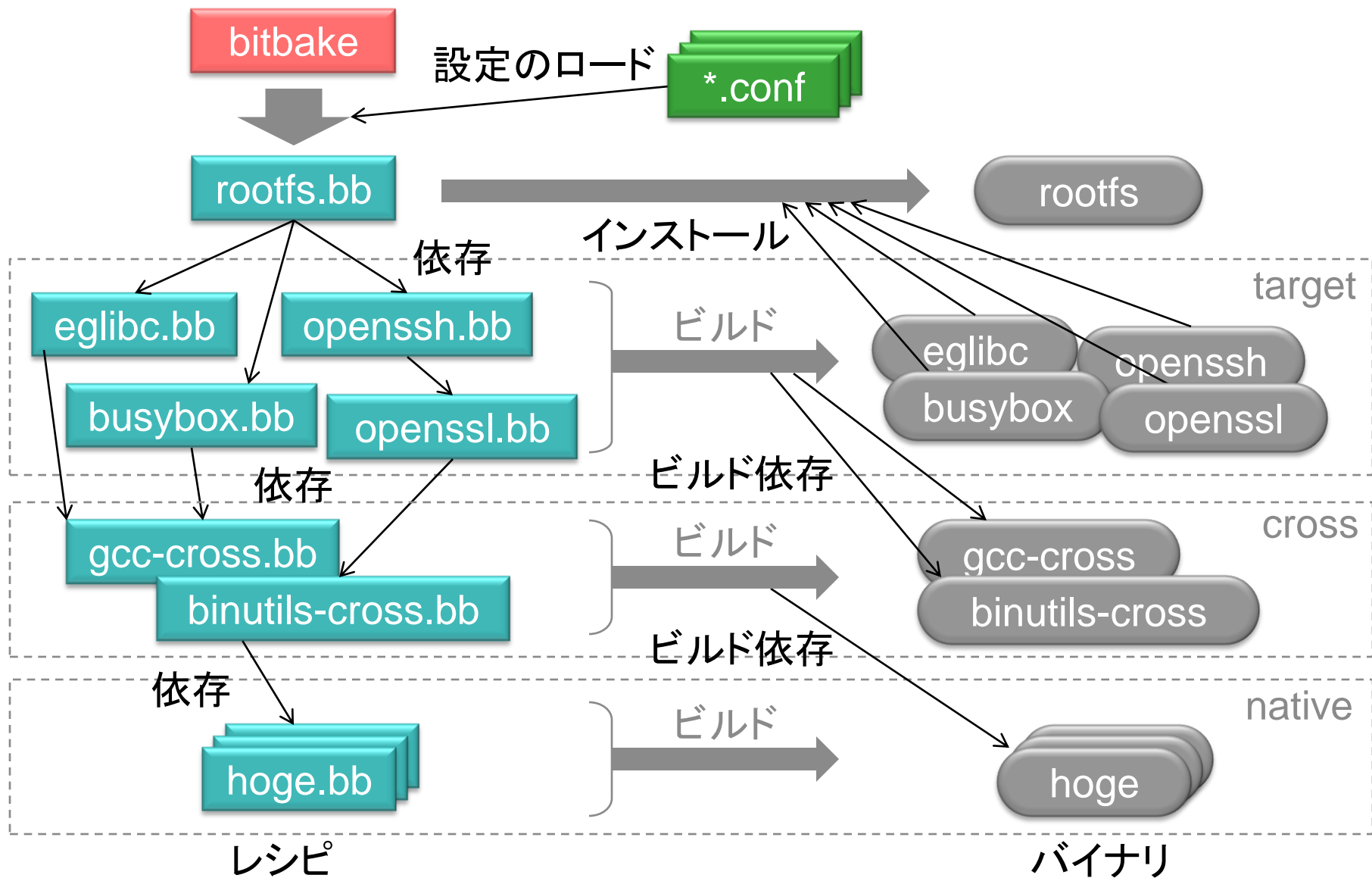
- ビルドツール(pythonスクリプト)
- レシピファイルを解析し、ビルドアクションを実行
- 実行例) `$ bitbake busybox`

■ 設定ファイル (*.conf)

- ターゲットマシン別のビルド定義
 - アーキテクチャ指定(march等)、出力形式(ulmage等)
- ユーザ独自設定
 - ターゲットマシンの選択、並行ビルドの設定等



ビルドフロー



BSPによるターゲット依存部の分離

■ BSP

- ターゲット依存の設定を含むレシピセット
- 必要なBSPを再利用・組み合わせて独自環境をビルドできる

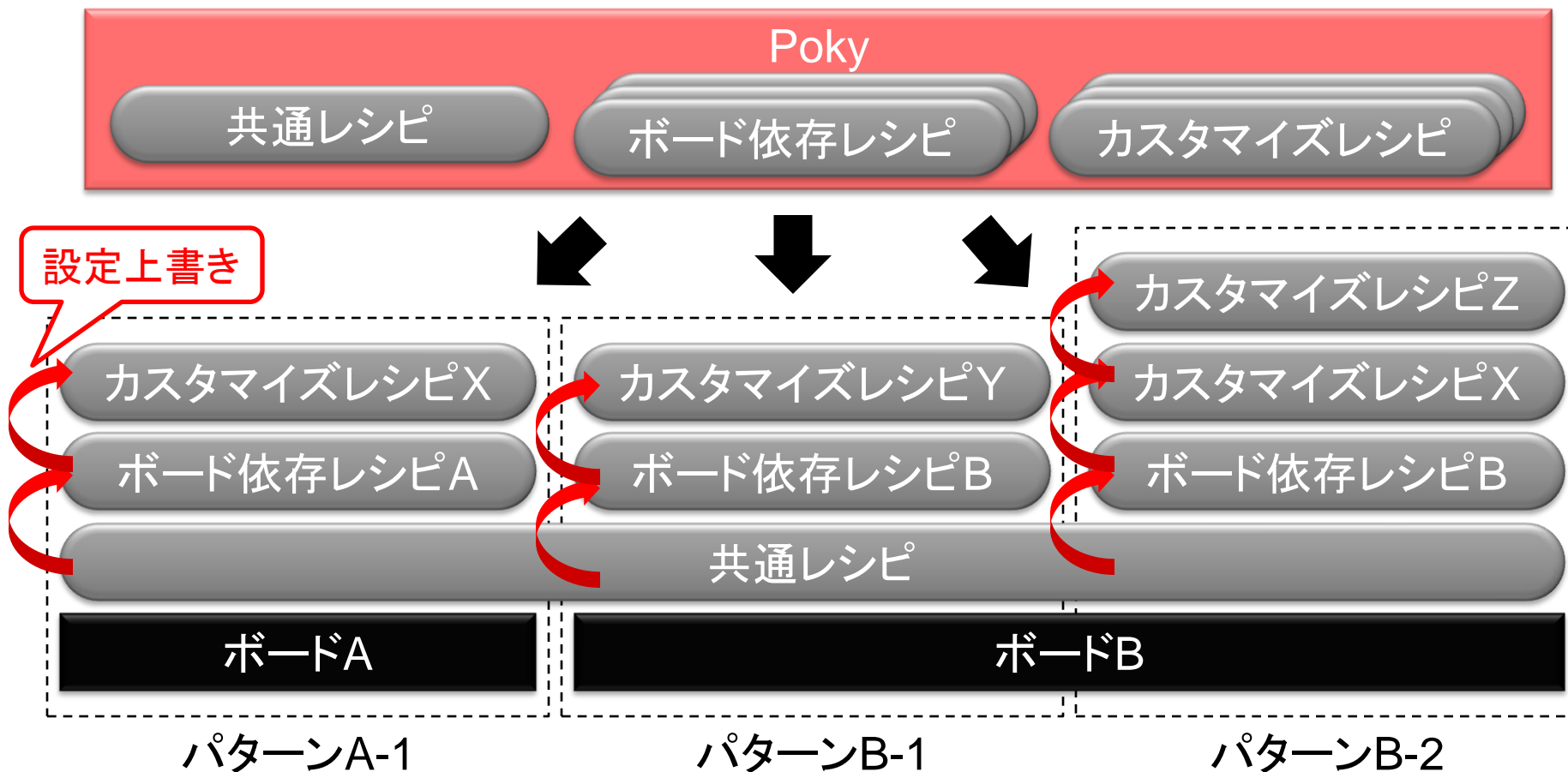
■ ターゲット依存設定の例

- ターゲットボード(H/W)依存の設定
 - カーネルソース、.config、イメージ形式(zImage, uImage, etc.)
 - ブートローダのデフォルトパラメータ(ロードアドレス等)
 - 静的デバイスファイル、コンソール名(ttyAMA, ttyO, etc.)
- 環境ごとにカスタマイズする設定
 - /etc以下の設定
 - rc, inittab, fstab
 - 各種デーモン設定
 - configureオプション(.config)の変更

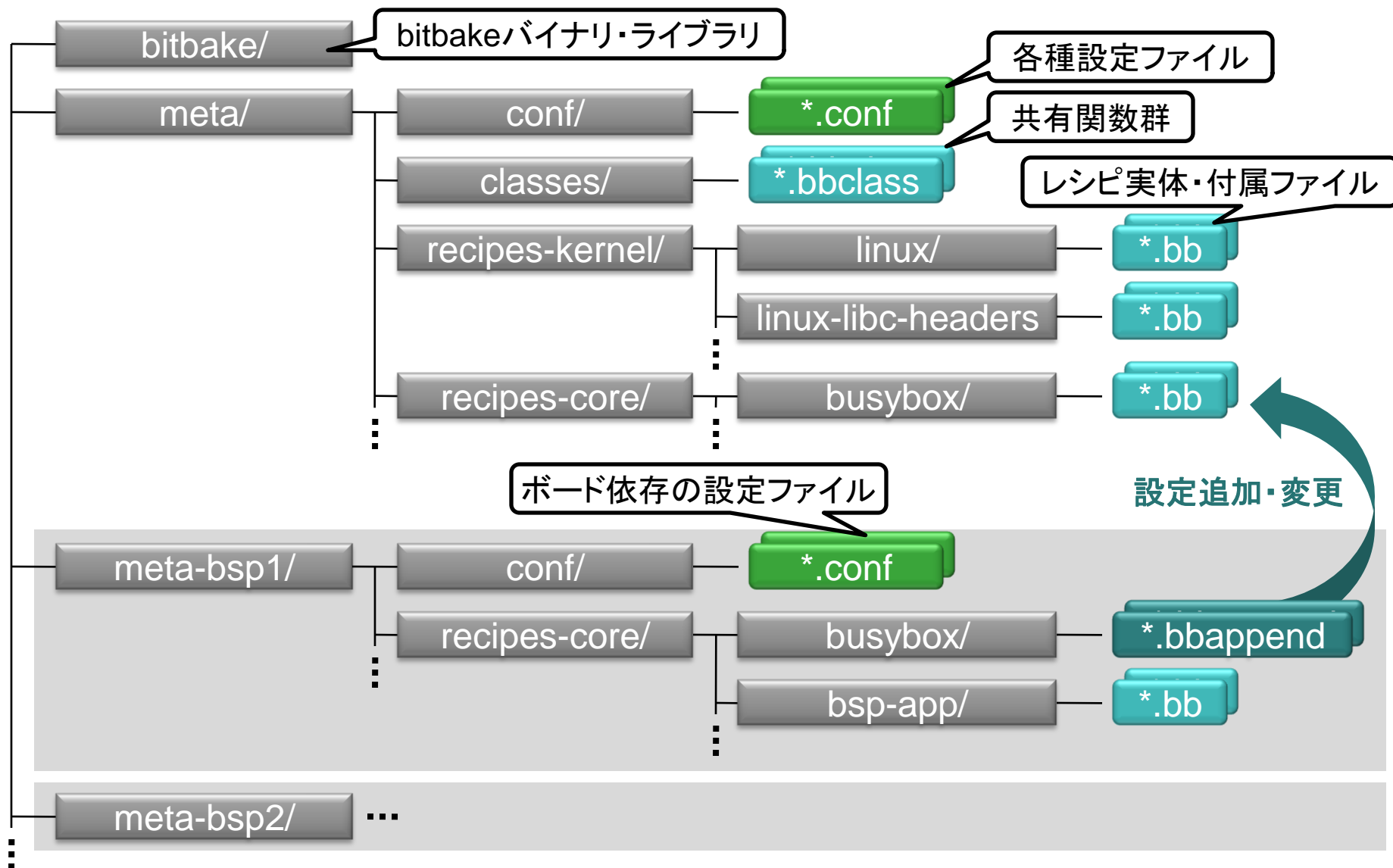
BSPによるターゲット依存部の分離

■ 全ボード共通レシピ + ターゲット依存レシピ (BSP) の構成

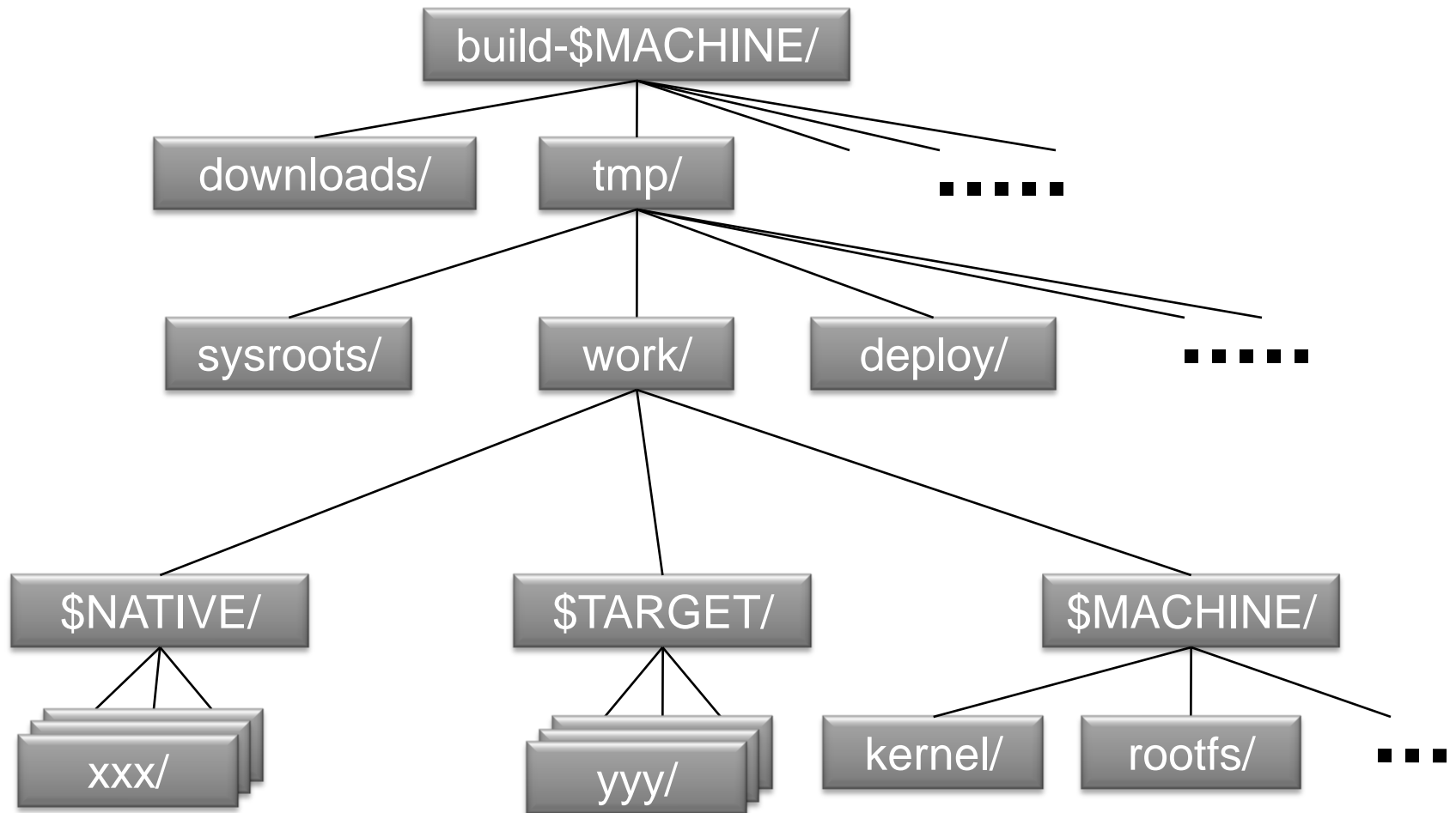
- レイヤ構造
- 各BSPが共通レシピに対し追加設定をオーバライド



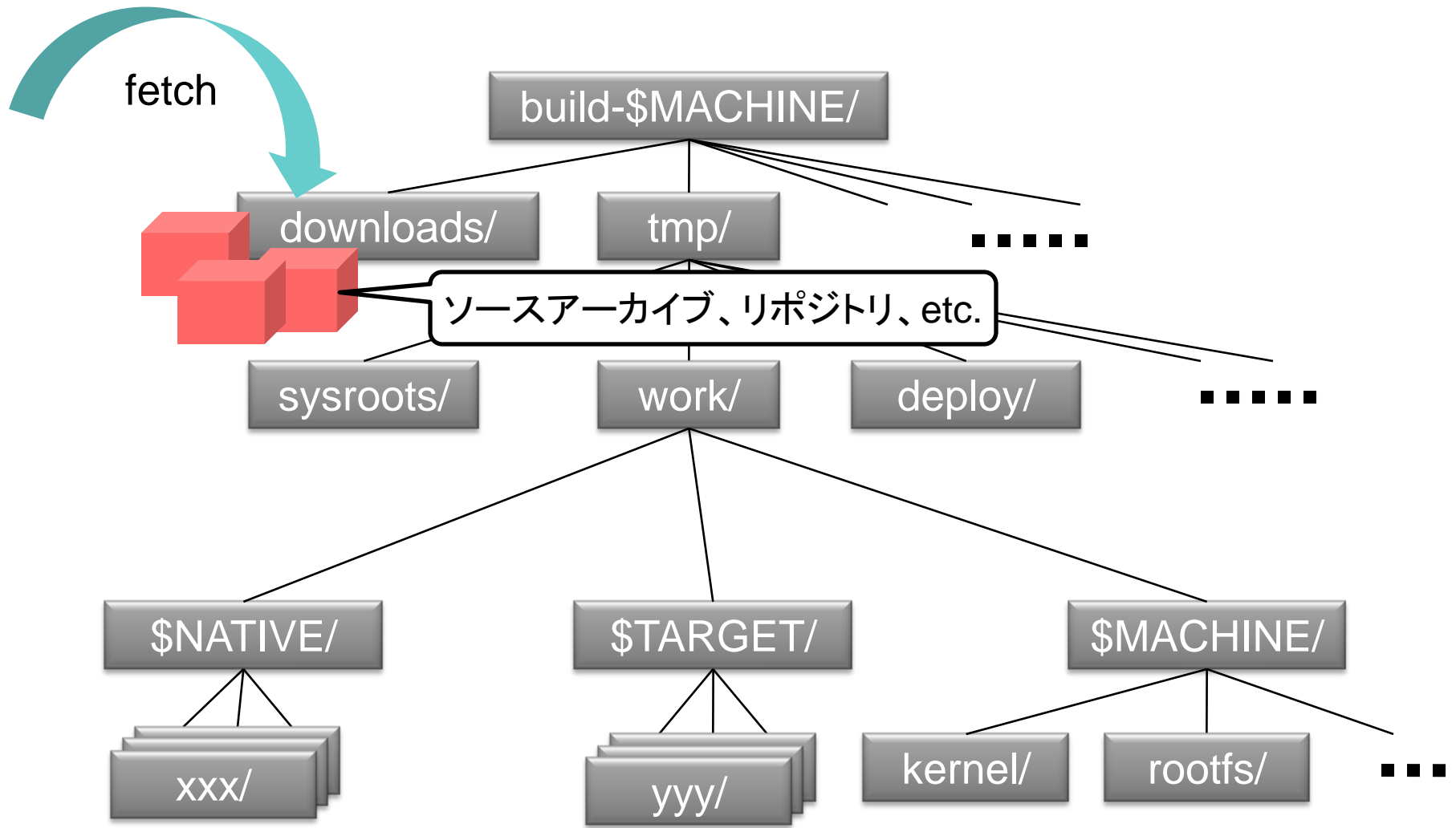
Pokyのディレクトリ構造



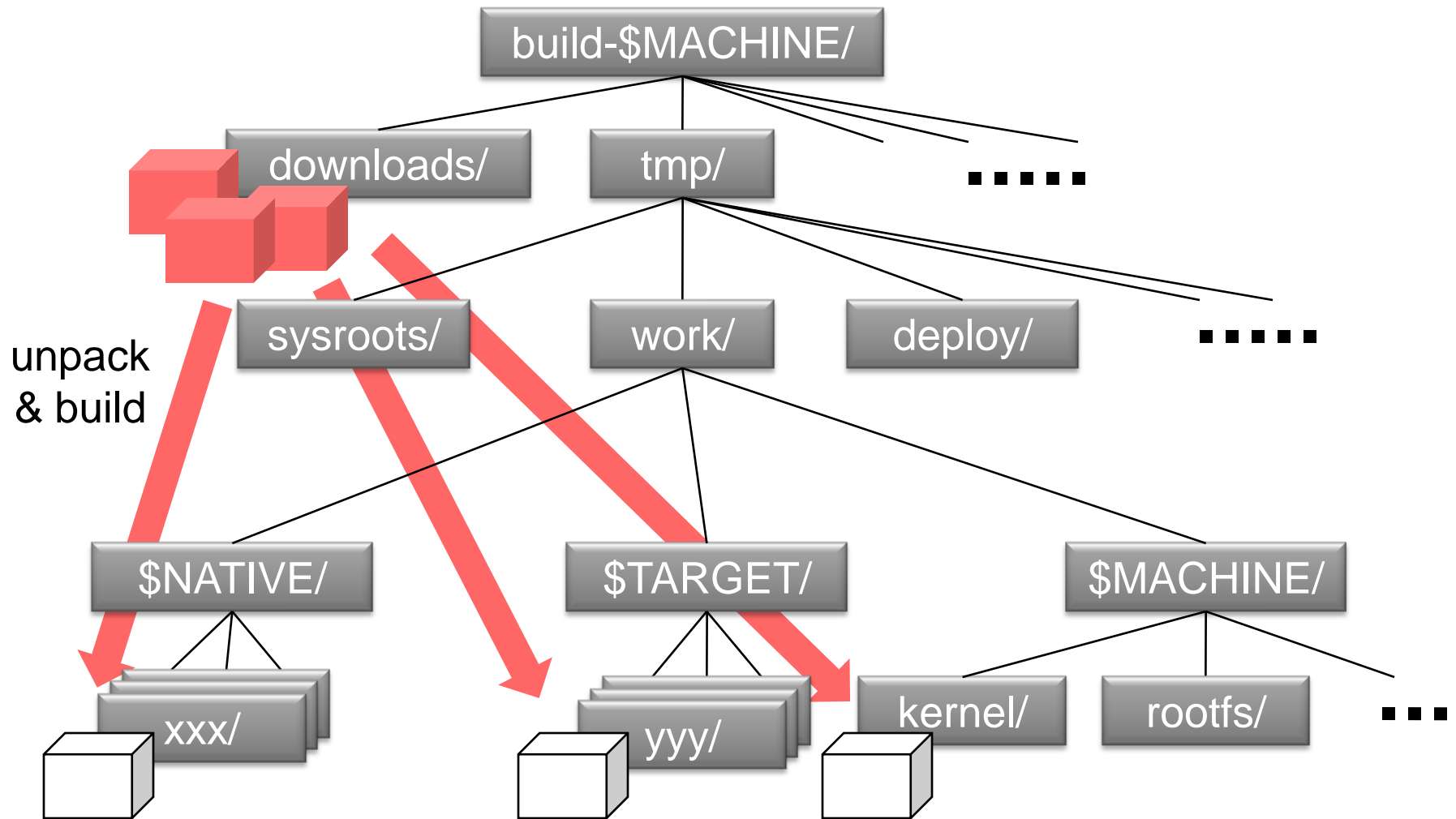
ビルドディレクトリの構造とビルドフロー



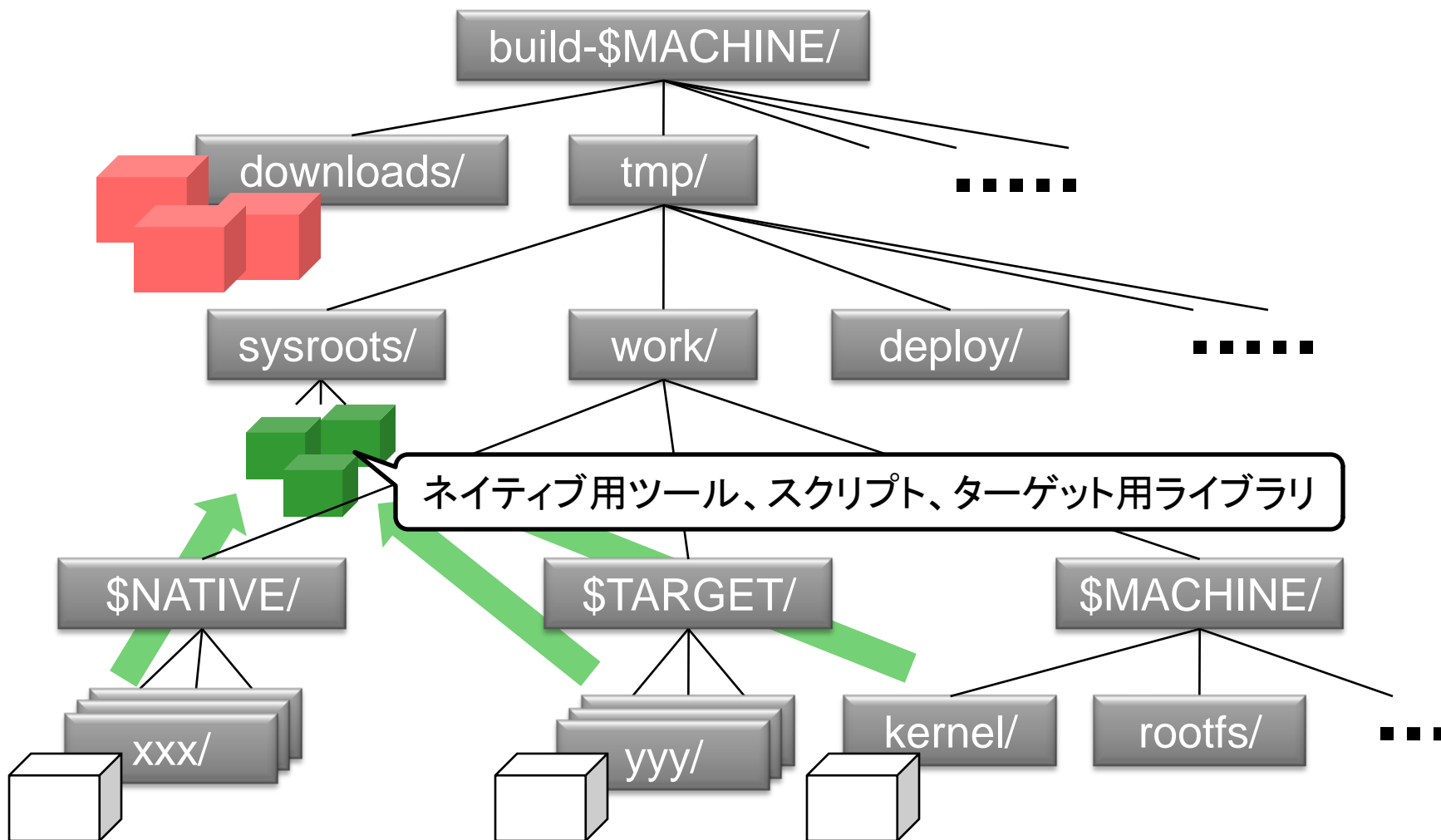
ビルドディレクトリの構造とビルドフロー



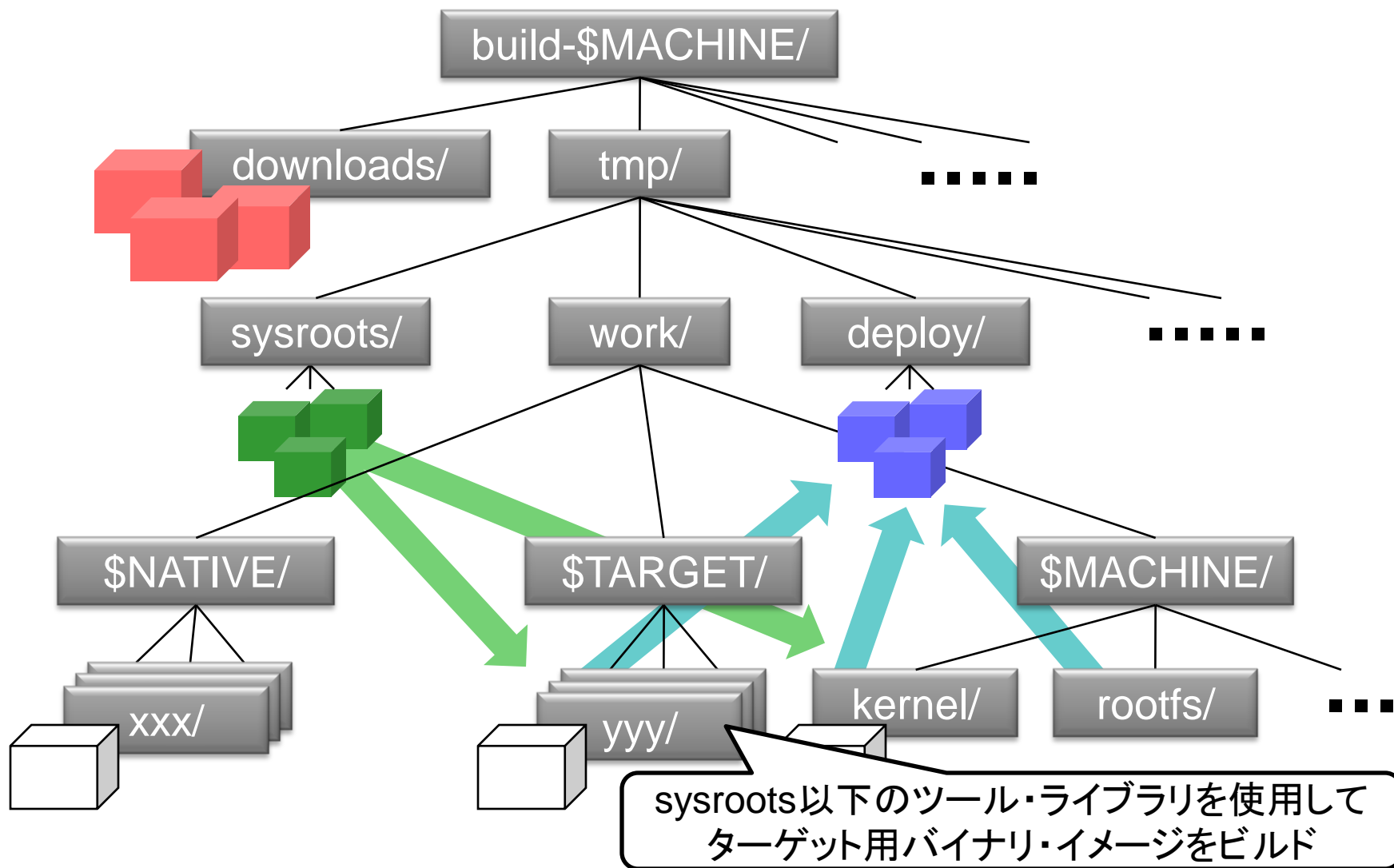
ビルドディレクトリの構造とビルドフロー



ビルドディレクトリ構造とビルドフロー



ビルドディレクトリ構造とビルドフロー



目次

■ Pokyの概要

- 全体像
- 構成要素
- ビルドフロー
- ディレクトリ構造

■ レシピの作成

- 記述方法
- 例)helloレシピ

■ Pokyをカスタマイズする

- バージョンアップ
- カーネルレシピの作成
- BSPの作成

■ 使用上の課題

レシピの作成

- ライセンス定義
 - タスク定義
 - ソースファイル定義
 - ビルド依存定義
 - パッケージングの設定
-
- 例: `hello.bb`

レシピの作成 – ライセンス定義

```
LICENSE = "GPLv2"
```

```
LIC_FILES_CHKSUM = "file://COPYING;md5=1a2b3c..."
```

- **LICENSE:ライセンス名**

- **LIC_FILES_CHKSUM: ファイル名 + md5**

- COPYING、LICENSE等

- .cや.hの一部を同時に参照する場合もあり

- フォーマット:

- ```
file://FILENAME;beginline=n1;endline=n2;md5=CHECKSUM
```

# レシピの作成 – タスク定義

---

- bitbakeはタスク定義に従ってビルドコマンドを実行
- 基本タスク
  1. do\_fetch: ソースのダウンロード(アーカイブ、gitリポジトリ、等)
  2. do\_unpack: ソースを展開
  3. do\_patch: ローカルパッチの適用
  4. do\_configure: `configure`実行
  5. do\_compile: `make`実行
  6. do\_install: `make install`実行
  7. . . .
- 規定タスクは定義済み、必要に応じて上書き
- タスク間には依存関係あり
- 独自タスクを追加定義可能

# レシピの作成 – タスク定義(例)

```
do_patch() {
 if ["${PV}" -ge "3"]; then
 patch -p1 < ${WORKDIR}/extra.patch
 fi
}
```

別処理で上書き(条件を追加したい場合等)

```
do_configure() {
 :
}
```

空処理で上書き(configureが不要な場合等)

```
do_install_append() {
 install -d ${D}/${bindir}
 install -m 0755 ${S}/extrafile ${D}/${bindir}
}
```

prepend/appendで規定関数前/後に  
処理を追加可能

```
addtask hoge before do_configure after do_patch
do_hoge() {

}
```

独自タスクを追加

# レシピの作成 – ソースファイル定義

```
SRC_URI = " ¥
file://localfile.patch ¥
http://www.server.com/archive.tar.gz ¥
git://git.server.com/repo.git;protocol=git;tag=xxx ¥
"
```

## ■ SRC\_URI:ソースファイルの一覧

- プロトコル: file, http, git, svn, cvs, etc.

## ■ 規定タスクの一部はSRC\_URIを見て自動で処理を実行

- do\_fetch: プロトコルを見てwget, gitなどのfetchコマンドを自動実行
- do\_unpack: 拡張子を見て展開コマンドを自動実行(tar, git checkout)
- do\_patch: \*.patch、\*.diffなどを自動適用

# レシピの作成 – ビルド依存定義

---

```
DEPENDS = "foo bar baz"
```

- **DEPENDS: 事前にビルドしておく必要のあるレシピ一覧**
  - ビルドで必要なライブラリなどを事前に生成
  - .debのBuild-dependsフィールドに該当

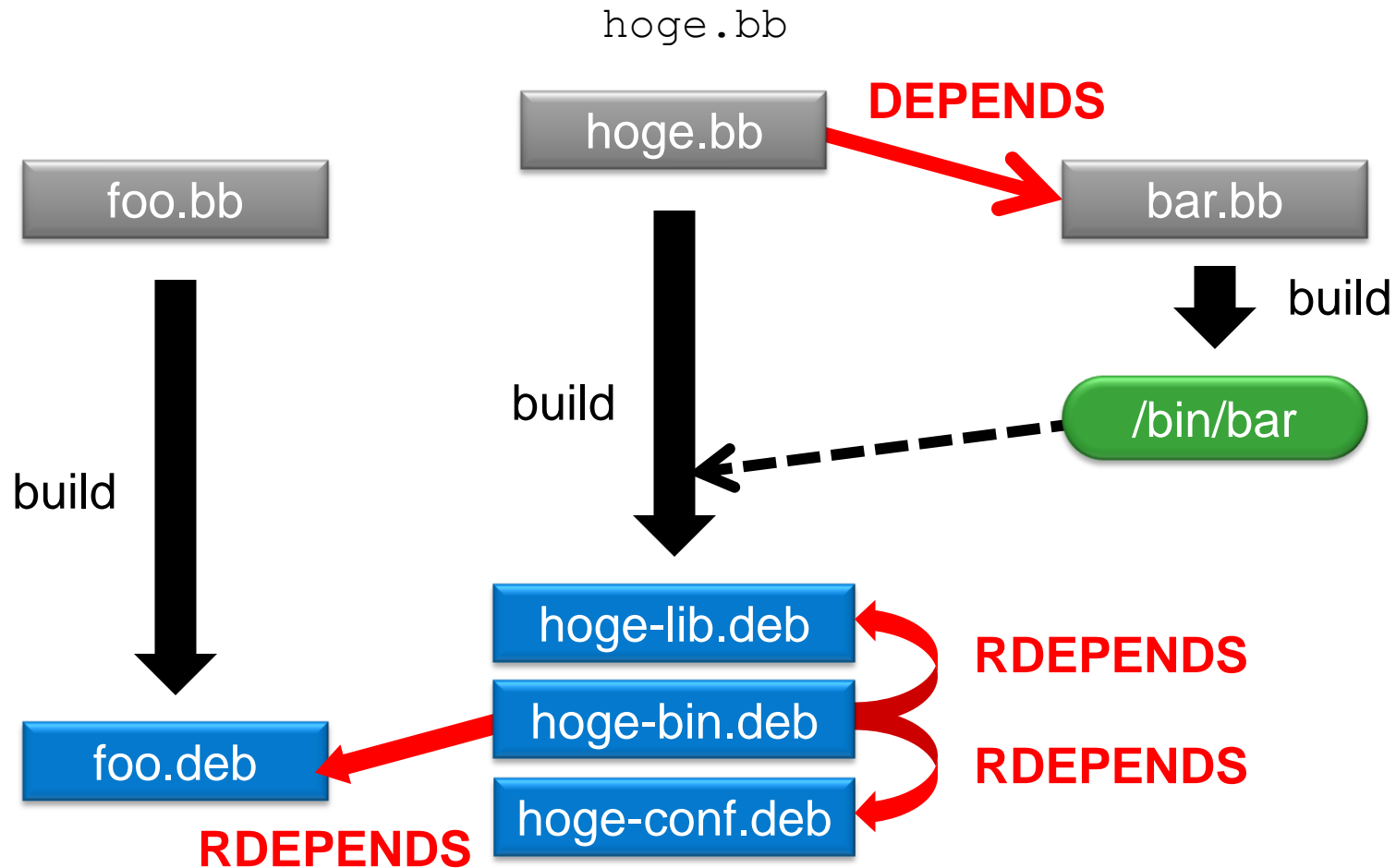
# レシピの作成 – パッケージングの設定

```
PACKAGES = "hoge-bin hoge-lib hoge-conf"
FILES_hoge-bin = "${bindir}/*"
FILES_hoge-lib = "${libdir}/*"
FILES_hoge-conf = "${sysconfdir}/*"
RDEPENDS_hoge-bin = "hoge-lib hoge-conf"
```

- ビルド済みファイルはPoky内部でパッケージングされる
  - パッケージ形式: .deb、.rpm、.ipk
- PACKAGES: 生成するパッケージ名一覧
- FILES\_xxx: パッケージに含めるファイル一覧
- RDEPENDS\_xxx: 依存パッケージ一覧
  - .debのDependsフィールドに該当

# レシピの作成 - 依存関係について

```
DEPENDS = "bar"
PACKAGES = "hoge-bin hoge-lib hoge-conf"
RDEPENDS_hoge-bin = "hoge-lib hoge-conf foo"
```



# レシピの作成(例) hello.bb

---

## ■ ソースコード

- 下記3ファイルを含む「hello.tar.gz」を作成

- hello.c

```
#include <stdio.h>
int main()
{
 printf("hello¥n");
 return 0;
}
```

- Makefile

```
default: clean hello
hello: hello.o
clean:
 rm -f hello *.o
```

- COPYING: ライセンス表記 (GPLv2)

# レシピの作成(例) hello.bb

## ■ hello.bb

```
LICENSE = "GPLv2"
LIC_FILES_CHKSUM = ¥
 "file://COPYING;md5=7c0d7ef03a7eb04ce795b0f60e68e7e1"

SRC_URI = "http://www.server.com/hello.tar.gz"

do_install() {
 install -d ${D}/${bindir}
 install -m 0755 ${S}/hello ${D}/${bindir}
}
```

### ■ bitbake内部変数

- \${S} (Source Directory): 展開済みソースディレクトリ
- \${D} (Destination Directory): インストール先ディレクトリ

### ■ \$ bitbake hello

→ ターゲットバイナリhelloがクロスコンパイルされる

# 目次

---

## ■ Pokyの概要

- 全体像
- 構成要素
- ビルドフロー
- ディレクトリ構造

## ■ レシピの作成

- 記述方法
- 例) helloレシピ

## ■ Pokyをカスタマイズする

- バージョンアップ
- カーネルレシピの作成
- BSPの作成

## ■ 使用上の課題

# Pokyをカスタマイズする

---

## ■ レシピの修正

- ソースコードをバージョンアップ

## ■ カーネルレシピの作成

- 自分で用意したカーネルを使う

## ■ BSPの追加

- 好きなボードで動くLinuxを作る

# レシピの修正

---

## ■ バージョン変更への要求

- 最新バージョンを使いたい
- 互換性のため古いバージョンを使いたい

## ■ Pokyのバージョン対応状況

- 1レシピが対応するバージョン数は少ない(ほぼ1:1)
- 目的のバージョンがない → レシピを追加

## ■ 修正手順

1. レシピのコピー & バージョン修正
2. バージョン依存部の修正
3. 付属ファイルの修正

# レシピの修正 ～ レシピのコピー＆バージョンの修正

```
files/
```

```
busybox.inc
```

```
busybox_1.18.5.bb
```

```
busybox-1.18.5/
```

```
busybox_1.20.2.bb
```

```
busybox-1.20.2/
```

バージョン依存のレシピ／付属ファイル

コピー＆バージョン変更

meta/recipes-core/busybox/

## ■ busybox.inc

- バージョンを含まないファイルは変更不要

## ■ busybox\_\${PV}.bb

- バージョン依存のレシピ記述（要修正）

## ■ busybox-\${PV}

- バージョン依存のパッチ、defconfig等（要修正）

# レシピの修正 ～ バージョン依存部の修正

```
require busybox.inc
PR = "r1"
```

```
SRC_URI = " ¥
http://www.busybox.net/downloads/busybox-${PV}.tar.bz2;name=tarball ¥
file://udhcpscript.patch ¥
...
file://defconfig"
```

自動で更新

```
SRC_URI[tarball.md5sum] = "...
SRC_URI[tarball.sha256sum] = "...
```

手動で更新

meta/recipes-core/busybox/busybox\_1.20.2.bb

- 内部変数\${PV}はファイル名から自動更新される
- ハードコードされた値・処理は手動で修正
  - checksum
  - configure／makeオプション
  - インストール処理
- 注意点
  - 他のレシピとの依存関係、互換性を維持する必要がある
  - ベースにしたバージョンからの差が大きいほど問題

# レシピの修正 ～ 付属ファイルの修正

---

## ■ 付属ファイル

- パッチ
  - Poky用の修正(パスなど)
  - バグfix
- defconfig
- 設定ファイル(\*.conf)

## ■ パッチは修正コスト大

- busybox\_1.20.2の場合:2個のパッチを修正
- 数千行規模のパッチが当たらなくなることも...
- 複雑さ次第ではメンテナレベルのノウハウが必要(力技)

# カーネルレシピの作成

---

- Pokyにおけるカーネルのビルド
  - デフォルトではYocto Projectのリポジトリを使用
- 自分のカーネルをビルドしたい → カーネルレシピを自作
- BeagleBoard向けの独自カーネルを使う例
  - linux-libc-headers-mybeagle.bbを作成
    - カーネルヘッダだけをPoky内部にインストールするレシピ
    - linux-libc-headers-yocto\_git.bbをベースに作成
  - linux-mybeagle.bbを作成
    - ターゲット向けカーネルイメージをビルドするレシピ

# カーネルレシピの作成 ~ linux-libc-headers-mybeagle

```
LICENSE = "GPLv2"
LIC_FILES_CHKSUM = "file://COPYING;md5=d7810fab7487fb0aad327b76f1be7cd7"

INHIBIT_DEFAULT_DEPS = "1"
PROVIDES = "linux-libc-headers"
RPROVIDES_${PN}-dev = "linux-libc-headers-dev"
RPROVIDES_${PN}-dbg = "linux-libc-headers-dbg"
DEPENDS += "unifdef-native"

SRC_URI = "git://localhost/linux.git;protocol=git"
SRCREV = "v3.3.7-mybeagle"
S = "${WORKDIR}/git"
KARCH = "arm"

do_configure() {
 oe_runmake allnoconfig ARCH=${KARCH}
}
do_compile () {
 :
}
do_install() {
 oe_runmake headers_install ¥
 INSTALL_HDR_PATH=${D}${exec_prefix} ARCH=${KARCH}
}
BBCLASSEXTEND = "nativesdk"
```

gitリポジトリを指定

チェックアウト対象のブランチ／タグを指定

recipes-kernel/linux-libc-headers/linux-libc-headers-mybeagle.bb

# カーネルレシピの作成 ~ linux-mybeagle

```
LICENSE = "GPL"
LIC_FILES_CHKSUM = "file://COPYING;md5=d7810fab7487fb0aad327b76f1be7cd7"
```

```
inherit kernel
```

カーネルビルド用のタスク定義をインポート

```
SRC_URI = "git://localhost/linux.git;protocol=git"
SRCREV = "v3.3.7-mybeagle"
S = "${WORKDIR}/git"
```

linux-libc-headers-mybeagle.bbと同じ

```
do_configure_prepend() {
 cp ${S}/config.mybeagle ${WORKDIR}/defconfig
}
```

作成済みの.configを使う場合

recipes-kernel/linux/linux-mybeagle.bb

## ■ kernel.bbclass

- configure, compile, install, package, deployを定義
- \${WORKDIR}/defconfigが.configとして使用される

## ■ 動的にmenuconfigを実行 & 編集する

- bitbake -c menuconfig virtual/kernel

# BSPの追加

## ■ 特定ボード向けのLinux環境一式を作りたい → BSPを自作

- meta-beagleboard
- meta-cubox
- meta-raspberrypi
- etc.



## ■ BeagleBoard向け独自BSPを作る場合

1. meta-mybeagleを作成
  - meta-skeletonをベースに
2. BSP設定ファイルの作成
3. マシン設定の作成
4. レシピのオーバライド
5. レイヤ設定の修正



# BSPの追加 ～ BSP設定ファイルの作成

```
BBPATH := "${BBPATH}:${LAYERDIR}"
BBFILES := " ${BBFILES} ¥
${LAYERDIR}/recipes-*/*/*.bb ¥
${LAYERDIR}/recipes-*/*/*.bbappend ¥
"
```

```
BBFILE_COLLECTIONS += "mybeagle"
BBFILE_PATTERN_mybeagle := "^${LAYERDIR}/"
BBFILE_PRIORITY_mybeagle = "6"
```

```
MACHINE = "mybeagle"
```

BSP名

レシピ優先度

ターゲットマシン

meta-mybeagle/conf/layer.conf

- **BSP名**
- **レシピの優先度(BBFILE\_PRIORITY\_\*)**
  - 値の高いBSPに属するレシピが優先される
- **ターゲットマシンの決定(MACHINE)**
  - conf/machine/\${MACHINE}.confが使用される

# BSPの追加 ～ マシン設定の追加

```
require conf/machine/include/tune-cortexa8.inc
```

CPU設定

```
PREFERRED_PROVIDER_virtual/kernel = "linux-mybeagle"
```

```
PREFERRED_PROVIDER_linux-libc-headers = "linux-libc-headers-mybeagle"
```

```
KERNEL_IMAGETYPE = "uImage"
```

カーネル設定

```
IMAGE_FSTYPES += "tar.bz2 ext3"
```

```
SERIAL_CONSOLE = "115200 ttyO2"
```

rootfs設定

```
EXTRA_IMAGEDEPENDS += "u-boot"
```

```
UBOOT_MACHINE = "omap3_beagle_config"
```

```
UBOOT_ENTRYPOINT = "0x80008000"
```

U-Boot設定

```
UBOOT_LOADADDRESS = "0x80008000"
```

meta-mybeagle/conf/machine/mybeagle.conf

## ■ CPU設定

- march等のCPU依存の設定(conf/machine以下の既存ファイルを流用)

## ■ カーネル設定

- カーネルレシピ(virtual/kernel)にmybeagle用カーネルを指定
- 出カイメージの指定(zImage, uImage)

## ■ rootfs設定

- 出カイメージの指定(tar ball, ext3, jffs2, cpio, etc.)
- シリアルコンソール設定(inittabに反映)

## ■ U-Boot設定

- 使用するconfig、ロードアドレスの指定(u-bootレシピの内部変数)

# BSPの追加 ～ レシピのオーバライド

```
do_install_append() {
 sed -i "s|.*¥(PermitRootLogin¥) yes|¥1 no|g" ¥
 ${D}/${sysconfdir}/ssh/sshd_config
}
```

sshd\_configの修正

meta-mybeagle/recipes-connectivity/openssh/openssh\_5.8p2.bbappend

```
IMAGE_INSTALL += "task-core-ssh-openssh"
IMAGE_DEVICE_TABLES += "files/device_table-mybeagle.txt"
```

- ・opensshをrootfsに追加
- ・静的デバイスファイルの一覧表を指定

meta-mybeagle/recipes-core/images/core-image-minimal.bbappend

| #   | path          | type | mode | uid | gid | major | minor | start | inc | count |
|-----|---------------|------|------|-----|-----|-------|-------|-------|-----|-------|
|     | /dev          | d    | 755  | 0   | 0   | -     | -     | -     | -   | -     |
|     | /dev/console  | c    | 662  | 0   | 0   | 5     | 1     | -     | -   | -     |
|     | /dev/kmem     | c    | 640  | 0   | 15  | 1     | 2     | -     | -   | -     |
|     | /dev/mem      | c    | 640  | 0   | 15  | 1     | 1     | -     | -   | -     |
|     | /dev/null     | c    | 666  | 0   | 0   | 1     |       |       |     |       |
| ... |               |      |      |     |     |       |       |       |     |       |
|     | /dev/tty      | c    | 600  | 0   | 5   | 4     |       | 0     | 1   | 7     |
|     | /dev/ttyO2    | c    | 640  | 0   | 5   | 253   | 2     | -     | -   | -     |
|     | /dev/mmcblk0  | b    | 660  | 0   | 6   | 179   | 0     | -     | -   | -     |
|     | /dev/mmcblk0p | b    | 660  | 0   | 6   | 179   | 1     | 1     | 1   | 5     |

ボード依存のデバイスファイルを追加

meta-mybeagle/files/device\_table-mybeagle.txt

# BSPの追加 ～ レイヤ設定の修正 & ビルド

```
LCONF_VERSION = "4"
```

```
BBFILES ?= ""
```

```
BBLAYERS = " ¥
```

```
##COREBASE##/meta ¥
```

```
##COREBASE##/meta-yocto ¥
```

```
##COREBASE##/meta-mybeagle ¥
```

```
"
```

- ・BSPを追加(複数追加可)
- ・meta, meta-yoctoはデフォルト

meta-yocto/conf/bblayers.conf.sample

## ■ bblayers.conf.sample

- デフォルトのレイヤ設定
- BBLAYERS: 有効にするBSP一覧

## ■ ビルド

- bitbake core-image-minimal
- layer.conf、マシン設定、レシピのオーバライドが適用された状態でターゲット向けイメージがビルドされる

# 目次

---

## ■ Pokyの概要

- 全体像
- 構成要素
- ビルドフロー
- ディレクトリ構造

## ■ レシピの作成

- 記述方法
- 例)helloレシピ

## ■ Pokyをカスタマイズする

- バージョンアップ
- カーネルレシピの作成
- BSPの作成

## ■ 使用上の課題

# Pokyを利用する上での問題点

---

- 課題1： 時間
- 課題2： ディスク消費量

# ビルド時間

---

## ■ ビルド時間が長い

- 早くても30分程度
- マシン性能が悪いと半日かかることも

## ■ 主な原因

- native用ツール、クロスツールチェーンを含むフルビルド
- 依存関係の冗長性
  - configure次第では不要となる依存関係もある
  - ビルド依存先のレシピから連鎖的に拡大

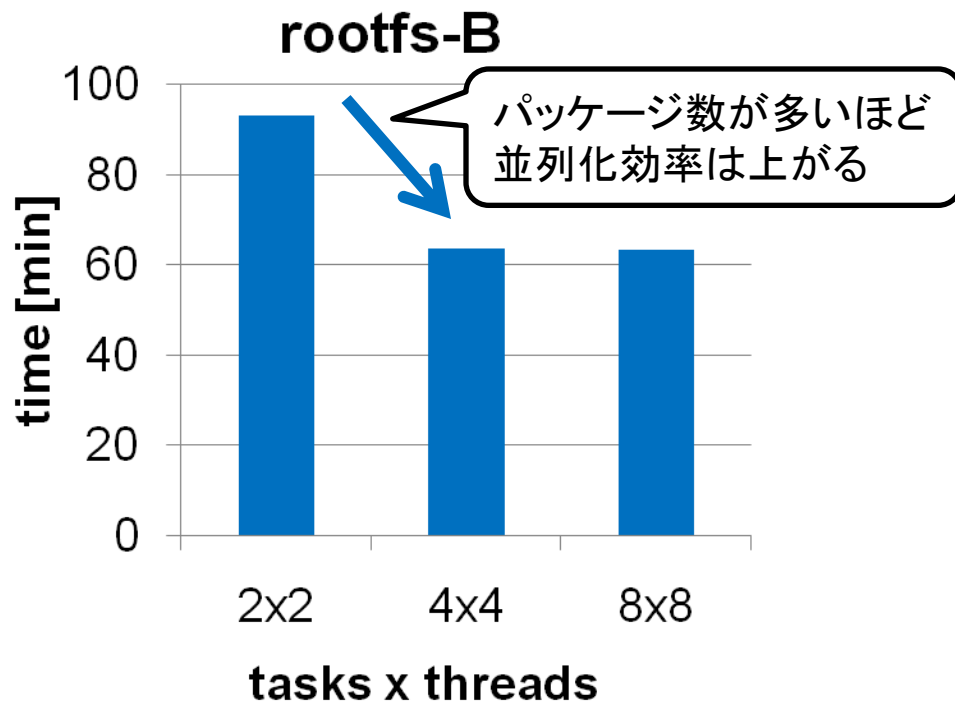
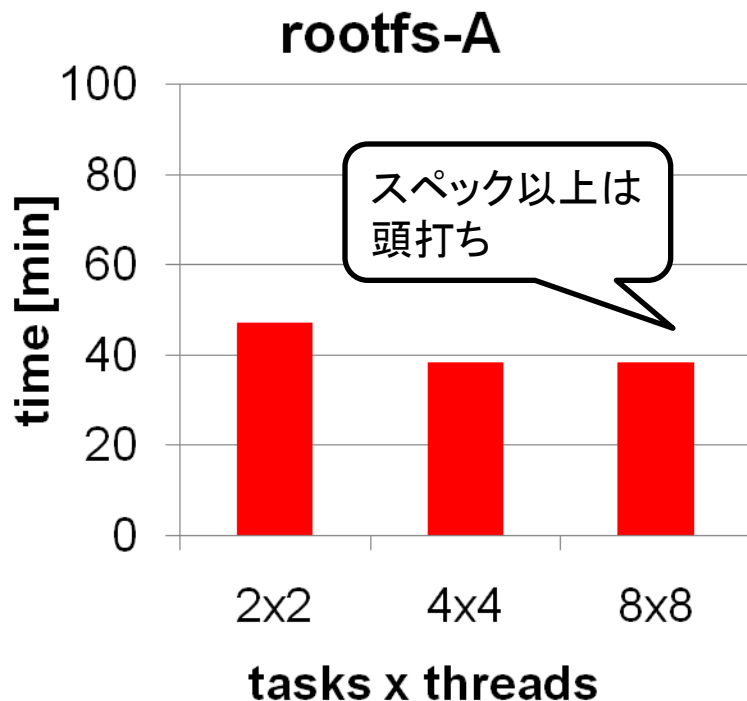


# ビルド時間の具体例

## ■ ビルド時間の計測

| ビルド対象    | レシピ数 | 概要                       |
|----------|------|--------------------------|
| rootfs-A | 84   | busyboxベースの最小rootfs+カーネル |
| rootfs-B | 145  | (A) + udev、sshd、Qtなど     |

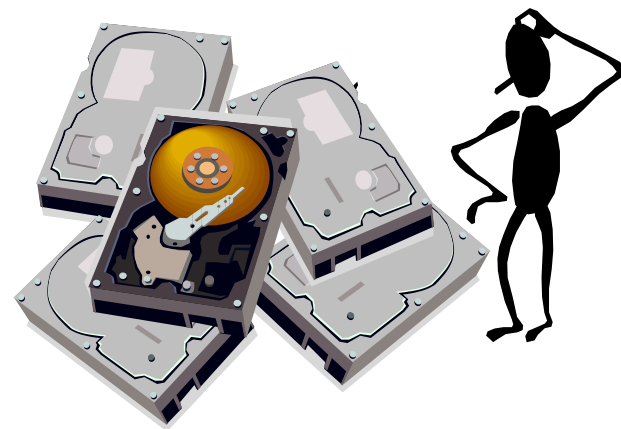
ビルド環境: Intel Xeon E3-1270(3.4GHz x4) / 24GB RAM / 2TB 7200rpm HDD



# ディスク消費量

## ■ ビルドディレクトリがディスクを圧迫

| ビルド対象     | レシピ数 | ビルド後の容量 |
|-----------|------|---------|
| rootfs-A  | 84   | 11GB    |
| rootfs-B  | 145  | 19GB    |
| busyboxのみ | 55   | 8.9GB   |



## ■ 主な原因

- native用ツール、クロスツールチェーンを含むフルビルド(前述)
- 依存関係の冗長性(前述)
- ビルド過程のファイルがすべて残っている
  - ダウンロードファイル、展開後のソース、バイナリ、パッケージファイル

## ■ 対策

- 他のレシピから参照されない構築後のファイルを自動削除
- ターゲット設定に完全に依存しない部分を共有したビルド方法
  - downloadデータ、native用ツールのワークディレクトリなど
- レシピ依存関係の改善

# まとめ

---

## ■ Pokyの概要説明

- 内部構造(ビルドシーケンス、ディレクトリ構造)
- サンプルレシピの記述方法
- Pokyを使用する主なメリット
  - クロスビルド環境を自動的に構築
  - BSPの活用による開発効率向上

## ■ Pokyのカスタマイズ方法

- バージョン変更
- カーネルレシピの作成
- BSPの追加

## ■ 課題

- ビルド時間
- ディスク使用量

## ■ Yocto Project

- <http://www.yoctoproject.org/>

## ■ Poky Reference Manual

- <http://www.yoctoproject.org/docs/1.1/poky-ref-manual/poky-ref-manual.html>
- Please read docs under version "1.1" (base of poky-skerlet)

## ■ OpenEmbedded User Manual

- <http://docs.openembedded.org/usermanual/usermanual.html>

## ■ BitBake User Manual

- <http://docs.openembedded.org/bitbake/html/>